Basics of Using `vp`, a Value-Pasing Front End for the Concurrency Workbench

Susan Older

DRAFT: November 6, 2003

Abstract

These notes describe some of the basics of using Glenn Bruns' `vp` [1], which is a value-passing front-end for the Concurrency Workbench [3]. The official documentation is available at http://www.bell-labs.com/user/grb/vp/vp.html, but it is fairly cryptic to the uninitiated. These notes are intended to fill in some of the gaps.

1 Running `vp`

An overview of Glenn Bruns' value-passing front end for the Concurrency Workbench is available at http://www.bell-labs.com/user/grb/vp/vp.html.

In particular, that web page includes pointers to a (postscript) user's guide [1] and the definition of value-passing CCS [2].

Running `vp` on the ECS Unix system is very simple:

1. Create a file with your definitions (see the rest of this writeup for more details). Make sure that the extension is `.vp`. For the rest of these instructions, imagine that the file is `example.vp`.
2. Run the following command at the Unix prompt (notice the `.vp` extension is **not** included):

   ```
   vp example
   ```

3. The result (if all goes well) is a file `example.ccs` that contains code that the Concurrency Workbench can handle.

2 Basic Components of Syntax

2.1 Constants

Constants serve the purpose that constants usually serve in a programming language: they support abstraction and a more elegant coding style. They are especially useful for defining sets that will be used to index or parameterize actions and processes.

Here are some sample constant declarations:

```
const I = {1,2}
const n = 5
const A = {2,4,6}
```

Notice that, unlike CWB notation, these commands do not terminate with semi-colons.

2.2 Parameterized labels

Parameterized labels are defined using the following syntax:

```
label b(I)
```

The result of this command is to create a *channel* `b` that accepts/passes values from the set `I`.

The action corresponding to *receiving* a value on channel `b` is written as follows (where `x` is any variable):

```
b(x)
```

The result of this command in a process description is to instantiate the variable `x` with the value received.

In contrast, the action corresponding to *passing* a value on channel `b` is written as follows (where `e` is any valid expression):

```
'b(e)
```

Thus, by using a *co-name* (i.e., something preceded by the CWB `'`), we indicate an output/passing action.

For example, the agent `Negate` continually receives a bit value on channel `a` and spits out its negation on channel `a`:

```
const Bit = {0,1}

label a(Bit)

agent Negate = a(w).'a(1-w).Negate
```

Again, notice the lack of a semi-colon.

If the above code were in a file called `Test.vp`, the result of executing the Unix command `vp Test` would be a file called `Test.ccs`, whose contents are as follows:

```
agent Negate = a%0%.'a%1%.Negate + a%1%.'a%0%.Negate;
```

2.3 Indexed labels

Indexed labels are defined using the following syntax:

```
label d_I
```

There is a subtle distinction between indexed labels and the previously mentioned parameterized labels.

The result of this command is to create multiple events, two for each element of the set `I`. For example, if `I` has been defined to be the set $\{1, 2\}$ (as above), then the result of this command is to create four actions (two names and two co-names):

$$\mathtt{d}_1, \mathtt{d}_2, {}'\mathtt{d}_1, {}'\mathtt{d}_2$$

Thus, the index is used to create a number of actions that are indexed by a set. However, they cannot be used in a generic way to pass values between agents. For example, running `vp` on the following code produces unexpected results:

```
const Bit = {0,1}

label a_Bit

agent Negate = a(w).'a(1-w).Negate
```

However, one can define the intended `Negate` agent explicitly as follows (of course, we could've done this directly in CWB as well):

```
const Bit = {0,1}

label a_Bit

agent Negate = a_0.'a_1.Negate + a_1.'a_0.Negate
```

The result (after running `vp`) is the following CWB-friendly file:

```
agent Negate = a%^0%.'a%^1%.Negate + a%^1%.'a%^0%.Negate;
```

Notice that the CWB-reprensentation of an *indexed* action is `a%^0%`, whereas the CWB-reprensentation of a *parameterized* action is `a%0%` (i.e., without the `^`).

2.4 Indexed *and* parameterized labels

We can combine these notations to create an indexed collection of channels. For example, consider the following code:

```
const Bit = {0,1}
const A = {2,4,6}

label a_A(Bit)

agent NegateA = a_2(w).'a_2(1-w).NegateA
              + a_4(w).'a_4(1-w).NegateA
              + a_6(w).'a_6(1-w).NegateA
```

Here, the agent `NegateA` is willing to receive a value on any one of three channels (`a_2`, `a_4` and `a_6`); its second step always involves output along the same channel.

2.5 Comments

Comments start with the characters `(*` and end with the characters `*)`, and they can span multiple lines. For example, the following is a valid comment:

```
(*********************************************************
   The mutual-exclusion problem:

     Two processes (P1 and P2) both have access to a shared
     resource. We want to coordinate access to that resource
     such that simultaneous access is prohibited.
 *********************************************************)
```

Likewise, here's another valid comment:

```
(* Here's a small one-line comment *)
```

3 Property Checking

Unfortunately, there does not seem to be a way to include μ-calculus properties in a `vp` file. Instead, they need to be included in a separate CWB file. As a further caveat, any properties in that file need to use the CWB syntax for parameterized and indexed actions.

My suggestion (and an approach I illustrate in the next section) is to create a CWB file that contains the desired properties. In addition, that file's first line should be a command to input the `.ccs` file generated by `vp`. For example, if my value-passing file is `example.vp`, then I create a file whose first line is:

```
input "example.ccs";
```

4 Putting it All Together

As an example, consider the Mutual-Exclusion problem discussed in lecture on the first day of class.

The file `Mutex.vp` appears in Appendix A; this file contains the value-passing code for the simple (but non-ideal) solution to the mutual-exclusion problem. The result of translating this file via `vp` is the file `Mutex.ccs`, given In Appendix B.

Finally, Appendix C contains the file `Mutex.cwb`, which includes the desired properties of the system. Note how the modal properties are expressed in terms of (ugly) actions such as `nonCrit%^1%`. Note also how this file includes the generated file `Mutex.ccs`.

These files are also available off the course web page.

References

[1] Glenn Bruns. The value-passing translator user guide, December 1996. Available from http://www.bell-labs.com/user/grb/vp/vp.html.

[2] Glenn Bruns. A language for value-passing CCS, January 1998. Available from http://www.bell-labs.com/user/grb/vp/vp.html.

[3] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

A The file `Mutex.vp`

```
(*************************************************************************
 *************************************************************************
 *************************************************************************
 ****
 **** The following sequence of processes are adapted from the following
 **** source:
 ****
 ****   Harvey M. Deitel, __An Introduction to Operating Systems__,
 ****     Revised First Edition, Addison-Wesley, 1984, Chapter 4.
 ****
 ****
```

```
**** Page numbers refer to the pages in the Deitel text.
****
*************************************************************************
*************************************************************************
*************************************************************************)

(*************************************************************************
   The mutual-exclusion problem:

     Two processes (P1 and P2) both have access to a shared resource.
     We want to coordinate access to that resource such that simultaneous
     access is prohibited.

     To be explicit, we want the solution to satisfy the following
     constraints:

        (1)  Deadlock should not occur.

        (2)  Both processes cannot be in their critical section (ie,
             accessing the shared resource) at the same time.

        (3)  Livelock should not occur: that is, there should not ever
             be the case that the only actions possible henceforth are
             tau actions.

        (4)  No process should be indefinitely postponed from entering
             its critical section (i.e., if a process is waiting to
             enter its critical section, it will eventually enter its
             critical section).

 *************************************************************************)


(*************************************************************************
 *************************************************************************
  Version 1  (Figure 4.3, page 81)

    Each process P_i (i=1,2) executes the following basic code, where
    the shared variable pnum is used to coordinate access:

       while true do
          do_non_critical_stuff;
          while pnum = (3-i) do skip;
          enter_critical_section_i;
          leave_critical_section_i;
          pnum := 3-i;
```

```
    Moreover, pnum is initially set to 1.

 ****************************************************************************
 ****************************************************************************)

(*** Processes are indexed by the set I = {1,2} ***)
const I = {1,2}

(*** Process actions are all indexed by their identity ***)
label nonCrit_I, wait_I, enter_I, leave_I

(*** In contrast, the variable Pnum's actions are **parameterized** by
 *** the set I.
 ***
 *** Why the difference?  A process should be able to read the variable,
 *** and see what its value is.
 ***)

label setPnum(I),getPnum(I)

(****************************************************************************
 ***
 *** Implementing the shared variable Pnum
 ***
 *** The variable pnum is represented by process Pnum(i): it can output
 *** its current value or update its current value.
 ***
 ****************************************************************************)

agent Pnum(i:I) = setPnum(j).Pnum(j) + 'getPnum(i).Pnum(i)

agent V1Proc(i:I) = nonCrit_i. wait_i. V1Wait(i)
agent V1Wait(i:I) = getPnum(j).
                        if j=3-i then wait_i.V1Wait(i)
                                 else enter_i.leave_i.'setPnum(3-i).V1Proc(i)



(** Note that the variable pnum is initially set to 1  **)
agent System1 = (V1Proc(1) | V1Proc(2) | Pnum(1))\{setPnum,getPnum}
```

B The file `Mutex.ccs`

```
agent Pnum%1% = setPnum%1%.Pnum%1% + setPnum%2%.Pnum%2% + 'getPnum%1%.Pnum%1%;
agent Pnum%2% = setPnum%1%.Pnum%1% + setPnum%2%.Pnum%2% + 'getPnum%2%.Pnum%2%;
agent V1Proc%1% = nonCrit%^1%.wait%^1%.V1Wait%1%;
```

```
agent V1Proc%2% = nonCrit%^2%.wait%^2%.V1Wait%2%;
agent V1Wait%1% = getPnum%1%.enter%^1%.leave%^1%.'setPnum%2%.V1Proc%1% + getPnum%2%.wait%^1%.V
agent V1Wait%2% = getPnum%1%.wait%^2%.V1Wait%2% + getPnum%2%.enter%^2%.leave%^2%.'setPnum%1%.V
agent System1 = (V1Proc%1% | V1Proc%2% | Pnum%1%)\{setPnum%1%,setPnum%2%,getPnum%1%,getPnum%2%
```

C The file `Mutex.cwb`

```
** load the (translated) value-passing file

input "Mutex.ccs";

***
***  Now define the desired properties
***
prop Always(Phi) = max (Z. Phi & [-]Z);
prop Eventually(Phi) = min (Z. Phi | (<->T & [-]Z));

***
***  Property (1):  Deadlock Freedom
***
***  This could also be defined as:
***
***         prop DeadlockFree = Always (<->T);

prop DeadlockFree = max (Z. <->T & [-]Z);

***
***  Property (2):  No simultaneous access
***
***  Whenever we see an enter_i event, we need to see a leave_i event before
***  we see another enter event.
***
***  That is: enter and leave events must cycle, and the first one we'll see
***  is an enter event.
***

set Leave = {leave%^1%,leave%^2%};
set Enter = {enter%^1%,enter%^2%};

prop NoSimultaneousAccess =
    max(Z. [Leave]F & [-Enter] Z
           & [Enter] max (Y.[Enter]F & [-Leave]Y & [Leave]Z));

***
***  Property (3):  No livelock
***
```

```
prop NoLivelock = Always (Eventually(<-tau>T));


***  Property (4):  No Indefinite Postponement
***
***  It turns out that we have to be careful here: the only way to be
***  guaranteed that a process isn't postponed is to make sure that
***  it'll be granted an opportunity by the scheduler.
***  Thus, we have to assume the existence of a fair scheduler.

prop NoIndefPostponement =
    ~FairScheduler |
  ( Always ([wait%^1%]Eventually(<leave%^1%>T))
  & Always ([wait%^2%]Eventually(<leave%^2%>T)));

**
** The property FairScheduler is really intended to say that every non-tau
** move that is ever enabled will eventually become disabled (and those that
** aren't enabled of course *stay* disabled).
**
** This property works as a notion of fairness only because, for processes
** P1 and P2, enabled actions never become disabled without actually occuring.
** (That is, the only choices involve choices between tau actions.)
**

set P1Moves = {enter%^1%,leave%^1%,nonCrit%^1%,wait%^1%};
set P2Moves = {enter%^2%,leave%^2%,nonCrit%^2%,wait%^2%};

prop FairScheduler =
    Always (Eventually([P1Moves]F))
  & Always (Eventually([P2Moves]F));
```